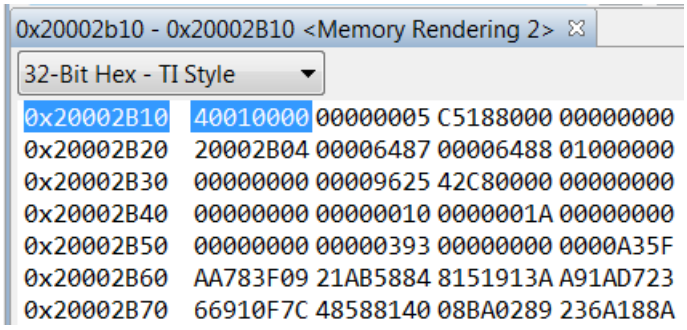
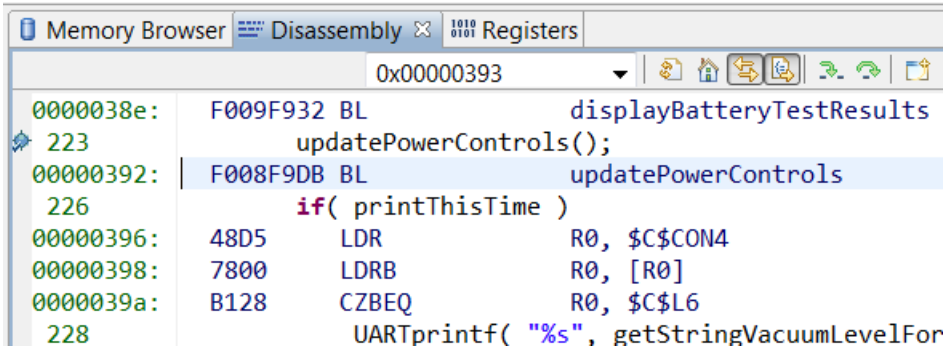
Debugging - tracing how got into ISR.docx

- http://e2e.ti.com/support/microcontrollers/tiva_arm/f/908/p/358677/1268902
  - A couple of great resources. That can help you backtrack out of the fault ISR and determine what line(s) of code are causing the issues. If you can use these to debug faults on a cortex M then you have learned a valuable skill that in my opinion puts you ahead of many folks on these forums. The best part is that it is not really that hard, just applied MCU debugging.
    - http://www.freertos.org/Debugging-Hard-Faults-On-Cortex-M-Microcontrollers.html
    - http://www.ti.com/lit/an/spma043/spma043.pdf
  - What you need to do is use the stack pointer and stack itself to figure out where the PC was before entering the ISR.
  - Before entering fault ISR a set of registers from the device are stored on the stack. Among those registers is the PC where the machine would return to if it were to return from the ISR.
  - This is all described in the document. I will try to summarize here.
  - In the debugger after hitting FaultISR().
    - Use the register window to find the address of the SP; this is top of stack.
    - ~~Use the memory browser to view the top of stack and about the next twelve 32 bit words under it. These are the most recent contents of the stack. Somewhere among those 12 words is the PC value you are looking for. Use the document to figure out where, i don't remember without looking myself.~~
    - Add 0x20 (32 decimal) to the SP to get the address on the stack which holds the return address from FaultISR() (the pushed value of the program counter). Use the Memory Browser to find the return address value stored on the stack.
    - Plug the return address PC value into your memory browser or disassembly window. This window will now point you to the approximate area of code that generated the fault ISR.

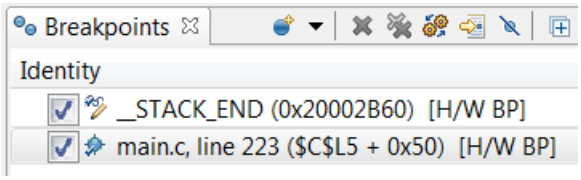
- Trying it (I am getting into FaultISR)
  - SP 0x20002B10. Right click and “View memory at value” to get this:



  - I know from stepping through my code earlier and watching the PC, that my code addresses for function main() are in the 0x300 range. The first number I see there which is in that range is 0x393. Copying that address to the clipboard and pasting it into the disassembly window (make sure to add a leading “0x”) gives me this:



- It is probably no coincidence that updatePowerControls() is a new function I just added that has not been tested yet with the current version of the PinMux file. I am probably trying to control a GPIO that isn't set up right.
- Setting a breakpoint on that line in the disassembly window makes it show up in the breakpoint window with the C source file name and line number:



- Double-clicking on that takes you to the source code line in the editor.
- Restarting the debug session and running to that breakpoint; it faulted before hitting the breakpoint.
- Starting again and single stepping. It faulted on the previous function call, displayBatteryTestResults(). Setting a breakpoint there.
- Starting debug session again. It gets to that breakpoint. Stepping in. It seems to fault on batteryGaugeCurtis840isUpdateNeeded(). That function must be accessing some resource that is now configured differently by PinMux.
  - It uses UART4, which is no longer configured. On HW_REV_00_B UART2 is set up to work with the Curtis gauge, so doing conditional compile to switch that out based on batteryGaugeCurtis840isUpdateNeeded(). Changing that fixed the problem; it no longer lands in FaultISR().

Modifying the ISR to make tracing easier:

```
// Enter an infinite loop.
// To trace back out of this ISR, use debugger to change the value
// of i to 0, then click the "Assembly Step Into" button several times.
//
volatile int i = 1;
while(i)
{}
```

What if can't trace out of ISR?

If had buffer overflow and hammered the stack, won't be able to step out of it, or even manually decode the return address from the stack. Now that FaultISR prints the tracepoints before going into infinite loop, may be able to figure out the last thing it did before crashing using tracepoints.